

Search space and brute force ...

If all you have is a hammer, everything looks like a nail

Search Space

- Ensemble of all solutions we have to choose from to get the optimal answer
- Often what is asked from us is not the solution but one of its properties

Search Space on an example : minimum pair

- Find the minimum pair on the list IE the pair with the minimum sum

0 5 8 6 -1 4 9 10 2



Search space : all the pairs

Make sense of your search space

Another representation :

Matrix where

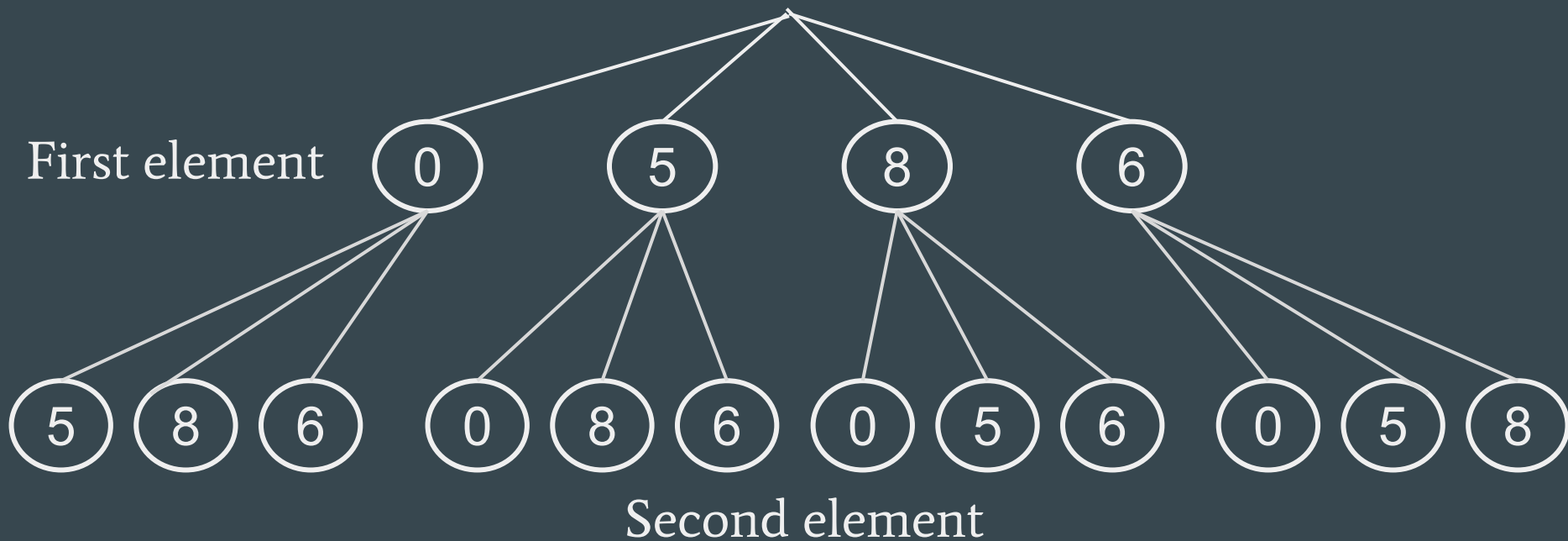
line : first element of the pair

Column : second element

Implementation : two for loops

	0	5	8	6
0	---	5	8	6
5	5	---	13	11
8	8	13	---	14
6	6	11	14	---

Make sense of your search space



Brute force - introduction

Example: I want to find all subsets of a set which sum is 42

I can:

- make a list of all subsets
- for each subset, calculate the sum and check if it is 42

Search space: all subsets of the array

Test: the sum is 42



*A brute force search has a
search space and a test*

Brute force - construct the search space

The python module `itertools` provides useful tools to construct the search space.

See <https://docs.python.org/3/library/itertools.html>

```
>>> list(product("ab", "cd"))
[('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')]
>>> list(permutations("abc", 2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
>>> list(combinations("abc", 2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
>>> list(combinations_with_replacement("abc", 2))
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
```

Note: we use `list()` because the return values are *lazy-evaluated* iterables

→ these are very useful to save memory

Quick tour of theses functions

product($q_1, q_2, \dots, [\text{repeat}=1]$)

```
product('ABCD', repeat=2)
```

```
AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
```

Can take multiple iterables and output the cartesian product between them

The repeat field indicate how many time do we take the iterables

Complexity : $O((\prod q_i)^{\text{repeat}})$

Where q_i is the size of each iterable

Quick tour of theses functions

permutation(p, [r=len(p)])

```
permutations('ABCD', 2)
```

```
AB AC AD BA BC BD CA CB CD DA DB DC
```

Take an iterable and create every permutation of length r

A permutation is an ordered (order is important) of element without repetition

Complexity : $O(\frac{p!}{(p-r)!})$ For example with r=2, the complexity is $p*(p-1)$

Where p is used as the size of the iterable p

Quick tour of theses functions

combinations(p, r)

```
combinations('ABCD', 2)
```

```
AB AC AD BC BD CD
```

Take an iterable and create every combination of length r

A combination is a non ordered of element without repetition, since it is non ordered, AB and BA are the same

Complexity : $O\left(\binom{p}{r}\right) = O\left(\frac{p!}{r!(p-r)!}\right)$

Where p is used as the size of the iterable p

Quick tour of theses functions

combinations_with_replacement(p, r)

```
combinations_with_replacement('ABCD', 2) | AA AB AC AD BB BC BD CC CD DD
```

Take an iterable and create every combination of length r with replacement

Same as combination, but we can repeat the same element multiple times

$$\text{Complexity : } O\left(\binom{p+r-1}{r}\right) = O\left(\frac{(p+r-1)!}{r!(p-1)!}\right)$$

Where p is used as the size of the iterable p

Brute force - let's solve our example

```
from itertools import combinations

arr = [int(n) for n in input().split()]

gen = (sub for size in range(len(arr))
        for sub in combinations(arr, size))

for sub in gen:
    if sum(sub) == 42:
        print(sub)
```

IN:

10 12 25 30 17 8 14 9 6

OUT:

(12, 30)

(25, 17)

(25, 8, 9)

(10, 12, 14, 6)

(10, 17, 9, 6)

gen is Python magic called a *generator expression*.

Ask me if you want to know more.

Why not to use brute force

There is a problem called *combinatorial explosion*

Cf https://en.wikipedia.org/wiki/Combinatorial_explosion

Tl; dr: the search space is often a lot bigger than the input size (e.g exponential)

Previous example: for an array of size n , there are $2^n - 1$ non-empty subsets...

Why ~~not~~ to use brute force

- Brute force is a good way to **test** if more complicated algorithms are correct (you compare their results on small sets)
- Sometimes you just don't know a better algorithm

But, can't we reduce the search space a little bit?

→ often you can use *backtracking* (see you next week)

Credits

Slides:

Louis Sugy, Arthur Tondereau